L16 - Error Handling

March 10, 2020

1 Handling Errors

Handling errors can be daunting. Lots of problems take place when developers forget to handle errors, and there are some applications where handling them systematically can be literally a life-or-death concern (aviation, space, nuclear, medical, etc.).

1.1 Typical errors

- programmer errors
  - incorrect implementation
    - your/someone else's code is wrong and doesnt' do what it says on the box
  - inappropriate object request
    - accessing off the end of an array
    - accessing an object that doesn't exist
  - inconsistent or inappropriate object state
    - your class extension messes up an object
- environmental or user errors
  - bad input from the user (filename, URL…)
  - network failure
  - missing files
  - no read permission on a file…

1.2 Fun with errors

Explore error situations with the `address-book` project.

1.2.1 Defensive programming

In a client server model, you need to decide how much your server should trust its clients - for proper arguments, etc.

- how much should a server check method calls?
- how should you report errors?

- how can a client anticipate failure?
- how should a client deal with a failure?

[2]:
```java
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;
import java.util.SortedMap;
import java.util.TreeMap;
import java.util.TreeSet;

/**
 * A class to maintain an arbitrary number of contact details.
 * Details are indexed by both name and phone number.
 *
 * @author David J. Barnes and Michael Kölling.
 * @version 2016.02.29
 */
public class AddressBook
{
    // Storage for an arbitrary number of details.
    private TreeMap<String, ContactDetails> book;
    private int numberOfEntries;

    /**
     * Perform any initialization for the address book.
     */
    public AddressBook()
    {
        book = new TreeMap<>();
        numberOfEntries = 0;
    }

    /**
     * Look up a name or phone number and return the
     * corresponding contact details.
     * @param key The name or number to be looked up.
     * @return The details corresponding to the key.
     */
    public ContactDetails getDetails(String key)
    {
        return book.get(key);
    }

    /**
     * Return whether or not the current key is in use.
     * @param key The name or number to be looked up.
     * @return true if the key is in use, false otherwise.
     */
    public boolean keyInUse(String key)
    {
        return book.containsKey(key);
    }

    /**
     * Add a new set of details to the address book.
     * @param details The details to associate with the person.
     */
    public void addDetails(ContactDetails details)
    {
        book.put(details.getName(), details);
        book.put(details.getPhone(), details);
        numberOfEntries++;
    }

    /**
     * Change the details previously stored under the given key.
     * @param oldKey One of the keys used to store the details.
     * @param details The replacement details.
     */
    public void changeDetails(String oldKey,
                              ContactDetails details)
    {
        removeDetails(oldKey);
        addDetails(details);
    }

    /**
     * Search for all details stored under a key that starts with
     * the given prefix.
     * @param keyPrefix The key prefix to search on.
     * @return An array of those details that have been found.
     */
    public ContactDetails[] search(String keyPrefix)
    {
        // Build a list of the matches.
        List<ContactDetails> matches = new LinkedList<>();
        // Find keys that are equal-to or greater-than the prefix.
        SortedMap<String, ContactDetails> tail = book.tailMap(keyPrefix);
        Iterator<String> it = tail.keySet().iterator();
        // Stop when we find a mismatch.
        boolean endOfSearch = false;
        while(!endOfSearch && it.hasNext()) {
            String key = it.next();
            if(key.startsWith(keyPrefix)) {
                matches.add(book.get(key));
            }
            else {
                endOfSearch = true;
            }
        }
        ContactDetails[] results = new ContactDetails[matches.size()];
        matches.toArray(results);
        return results;
    }

    /**
     * Return the number of entries currently in the
     * address book.
     * @return The number of entries.
     */
    public int getNumberOfEntries()
    {
        return numberOfEntries;
    }

    /**
     * Remove an entry with the given key from the address book.
     * @param key One of the keys of the entry to be removed.
     */
    public void removeDetails(String key)
    {
        ContactDetails details = book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
    }

    /**
     * Return all the contact details, sorted according
     * to the sort order of the ContactDetails class.
     * @return A sorted list of the details.
     */
    public String listDetails()
    {
        // Because each entry is stored under two keys, it is
        // necessary to build a set of the ContactDetails. This
        // eliminates duplicates.
        StringBuilder allEntries = new StringBuilder();
        Set<ContactDetails> sortedDetails = new TreeSet<>(book.values());
        for(ContactDetails details : sortedDetails) {
            allEntries.append(details).append("\n\n");
        }
        return allEntries.toString();
    }
}
```

[9]:
```
AddressBook ab = new AddressBook();

ab.addDetails( new ContactDetails("Sylvester", "111-111-1111", "Granny's␣
↪House"));

ab.addDetails( new ContactDetails("Wile E. Coyote", "222-222-2222", "Desert"));

System.out.println(ab.getNumberOfEntries());
```

```
2
```

[10]:
```
ab.removeDetails("Sylvester");
System.out.println(ab.getNumberOfEntries());
```

```
1
```

[11]:
```
ab.removeDetails("Tweety"); // not in the book
System.out.println(ab.getNumberOfEntries());
```

```
      ␣
↪---------------------------------------------------------------------------

       java.lang.NullPointerException: null

               at AddressBook.removeDetails(#20:1)

               at .(#36:1)
```

1.2.2 Argument values

Values of arguments are a big vulnerability for servers:

- constructors could start with invalid or dangerous state
- method arguments could corrupt the data in your object

Checking arguments can be an important way to defend your server

1.2.3 Error reporting

How do you want to report illegal arguments?

- to the user?
  - return a value from your method
- to the system log?
  - print to `System.err.println`

1.2.4 Client responses

- test return values
  - attempt recovery or at least don't blow up
- ignore the return value
  - if you can't avoid the error somehow
  - likely to lead to a crash
- use an *exception*

1.3 Exception Throwing

With exceptions we don't need special return values. If the client cannot ignore an error, control flows are interrupted in the program.

The server should just `throw` exceptions when methods are called badly. The client may `try ... catch` these.

1.3.1 To throw an exception

- an exception object is created: `new cleverExceptionName("...")`
- the exception is thrown: `throw cleverExceptionName`
- Javadoc: `@throws cleverExceptionType <useful description>`

[ ]:
```
// getDetails using an exception

public ContactDetails getDetails( String key )
{
    if ( key == null )
    {
        throw new IllegalArgumentException( "null key in getDetails")
    }

    return book.get(key); // normal operation of the key-fetcher
}
```

1.3.2 `Exception` inheritance

`Exception` inherits from `Throwable`; and `RuntimeException` inherits from `Exception`.

Checked exceptions

- a subclass of `Exception`
- checked by the compiler
- use for anticipated failures, where recovery may be possible

Unchecked exceptions

- a subclass of `RuntimeException`
- not checked by the compiler
- use for surprising failures, where there is no route to recovery
- terminates the program if not caught

1.3.3 Effects

The throwing method exits prematurely; no return value is generated. If the exception is caught, control returns to whoever called the throwing method.

Preventing object creation If an exception is built into a constructor, an invalid argument can prevent the construction of an object.

[17]:
```
public class Animal
{
    String name;
    String kind;

    public Animal( String name, String kind )
    {
        if ( name.trim().length() == 0 )
        {
            throw new IllegalStateException("Must specify a name");
        }

        if ( kind.trim().length() == 0 )
        {
            throw new IllegalStateException("Must specify a kind");
        }

        this.name = name;
        this.kind = kind;
    }
}
```

[18]:
```
Animal bb = new Animal("Bugs", "Bunny");
```

[23]:
```
Animal dd = new Animal("Daffy", "");
```

```
        ␣
↪---------------------------------------------------------------------------

        java.lang.IllegalStateException: Must specify a kind

                at Animal.<init>(#37:15)

                at .(#39:1)
```

`throws` clause Indicates that a method throws a checked exception. Add the clause `throws someKindOfException` to the method signature.

1.3.4 `try` statement

Clients that catch exceptions much protect the call with a `try` statement:

```
try
{
    do_something();
}
catch ( someException e )
{
    report_bad_thing_somehow();
    fix_your_stuff();
}
```

Once the call to `do_something()` in the `try` block throws an exception, execution passes to the `catch` block. If there isn't an exception, control passes through and the `catch` block never executes.

To catch multiple exceptions, add more `catch` blocks. Alternately you can multicatch (if you want to take the same action for different exceptions):

```
catch ( AnvilException | RocketException e )
{
    System.out.println("Ouch");
}
```

[22]:
```
try
{
    Animal dd = new Animal("Daffy", "");
}
catch( IllegalStateException e )
{
    System.out.println("Must be duck season");
}
```

```
Must be duck season
```

1.3.5 `finally` block

The code in the `finally` block executes after the actions in the `try` block, *whether or not an exception is thrown*, even if the `try` or `catch` blocks have a `return` statement. An *uncaught* (or *propagated*) exception will still exit via the `finally` block.

```
try
{
    do_something();
}
catch ( someException e )
{
    report_bad_thing_somehow();
    fix_your_stuff();
}

finally
{
    do_this_every_time_I_mean_it();
}
```

[24]:
```
try
{
    Animal pp = new Animal("Porky", "Pig");
}

catch ( IllegalStateException e )
{
    System.out.println("Ouch!");
}

finally
{
    System.out.println("That's all, folks");
}
```

```
That's all, folks
```

[25]:
```
try
{
    Animal pp = new Animal("", "Duck");
}

catch ( IllegalStateException e )
{
    System.out.println("Ouch!");
}

finally
```

```
{
    System.out.println("That's all, folks");
}
```

```
Ouch!
That's all, folks
```

1.3.6 Defining new exceptions

We can also declare a new exception type; this is just a new class that inherits `Exception`:

[29]:
```
public class mySuperDuperException extends Exception
{
    public mySuperDuperException ()
    {
        System.out.println("AWESOME");
    }
}
```

[34]:
```
throw new mySuperDuperException();
```

```
AWESOME


      ␣
↪---------------------------------------------------------------------------


        REPL.$JShell$50$mySuperDuperException: null

                at .(#55:1)
```